



Implementando o backend do projeto integrado

Depois de construirmos os ambientes onde nosso projeto rodará, começaremos, então, a sua implementação. Seguindo uma ordem lógica, que nos permita, inclusive, testar todas as funcionalidades, faremos primeiro o backend – tendo em mente que o banco de dados já foi implementado na configuração dos ambientes, ficando pendente apenas a parte de autenticação. Nesse sentido, ao longo deste módulo, confeccionaremos a API Restful com o Spring Boot. Antes de continuarmos, quero lembrá-lo do que você vai precisar para essa tarefa:

1. Banco de Dados Postgres (já configurado via Docker);

2. Script de criação das tabelas (já executado na subida do docker);

3. Complemento do script de criação de tabelas – incluiremos a parte de usuários/autenticação;

4. IDE para codificação – eu utilizarei o Spring Tools Suite. Você pode utilizar essa ou outra ferramenta à sua escolha;

5. Template do Projeto Spring, que pode ser gerado a partir do http://start.spring.io **(Acesso em 20/10/2022)** – inclusive, já geramos esse template no módulo anterior;



6. Estrutura da pasta dos códigos do backend – também fizemos isso no último módulo;

7. Swagger para testarmos a API, cuja dependência incluiremos em nosso pom.xml e também outra ferramenta, por exemplo, o Postman ou Insomnia, caso prefira testar a API fora do Swagger.

Bem, já podemos começar a colocar a mão na massa. Embora não exista, obrigatoriamente, uma camada pela qual devamos começar, eu gosto de iniciar pelas camadas mais baixas, ou seja, pelo mapeamento das tabelas, colunas e relacionamentos do banco de dados nas classes de Entidade. Após isso, faremos os repositórios. Contudo, vamos por partes. Abra o script contendo as tabelas do banco de dados (que será executado novamente quando formos subir o ambiente Docker) e adicione a seguinte instrução:

```
CREATE TABLE IF NOT EXISTS usuario (
        user_id         serial  NOT NULL  ,
        user_nome        varchar(255)   ,
        user_email       varchar(255)   ,
        user_password     varchar(255)   ,
        PRIMARY KEY ( user_id )
);
```

DDL para criação da tabela usuario

Com a tabela acima, nosso modelo de banco de dados estará completo. Feito isso, vamos agora criar as entidades. Antes, e pra facilitar pra você, veja abaixo o template da aplicação. Perceba que já incluí aqui a dependência do Actuator.

Project
Maven Project
Gradle Project
Language
Java
Kotlin
Groovy
Spring Boot
3.0.0 (SNAPSHOT)
3.0.0 (M5)
2.7.5 (SNAPSHOT)
2.7.4
2.6.13 (SNAPSHOT)
2.6.12
Project Metadata
Group com.projeto
Artifact integrado
Name integrado
Description API de Backend
Package name com.projeto.integrado
Packaging Jar War
Java 19 17 11 8
Dependencies
ADD DEPENDENCIES... CTRL + B
Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.
Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
PostgreSQL Driver SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
Spring Boot Actuator OPS
Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

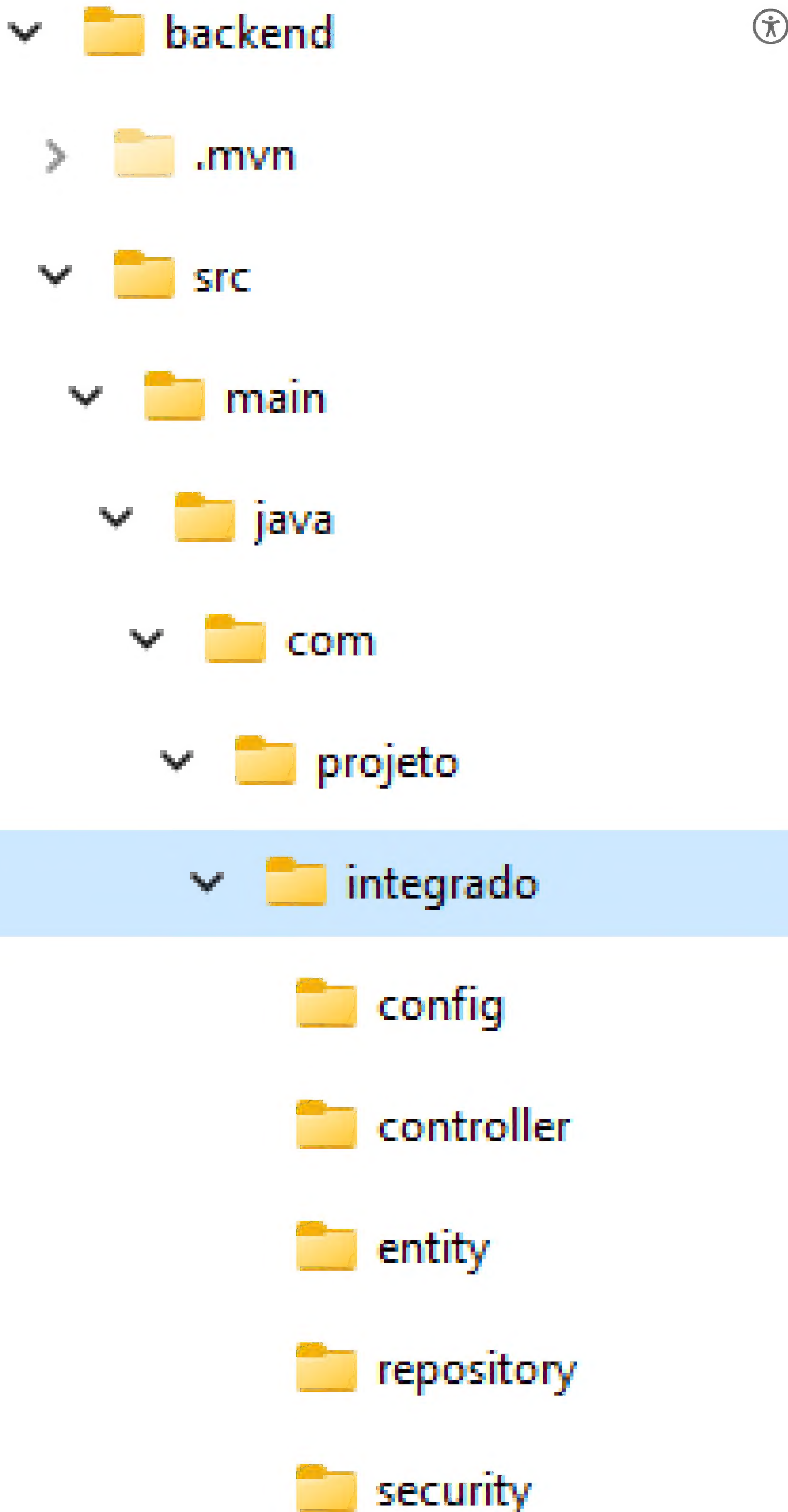
Figura 1: Template de Projeto Spring Boot

Ao gerar um novo zip ou aproveitando a estrutura do projeto que usamos no último módulo, e após ter incluído/conferido as dependências acima, edite o pom.xml e já inclua a dependência referente ao Springdoc e a java-jwt (usada na segurança da API). Nesse ponto, nosso pom.xml (apenas as dependências) deverá estar assim:



```xml
<dependencies>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-data-jpa</artifactId>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-security</artifactId>
	</dependency>
	<dependency>
		<groupId>com.auth0</groupId>
		<artifactId>java-jwt</artifactId>
		<version>3.19.2</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-web</artifactId>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-devtools</artifactId>
		<scope>runtime</scope>
		<optional>true</optional>
	</dependency>
	<dependency>
		<groupId>org.postgresql</groupId>
		<artifactId>postgresql</artifactId>
		<scope>runtime</scope>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-test</artifactId>
		<scope>test</scope>
	</dependency>
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-test</artifactId>
		<scope>test</scope>
	</dependency>
	<dependency>
	  <groupId>org.springframework.boot</groupId>
	  <artifactId>spring-boot-starter-actuator</artifactId>
	</dependency>
	<dependency>
		<groupId>org.springdoc</groupId>
		<artifactId>springdoc-openapi-ui</artifactId>
		<version>1.6.11</version>
	</dependency>
</dependencies>
```

Vamos conferir uma última questão antes de continuarmos. A estrutura de pastas, considerando os packages do projeto, deve estar assim:


backend
.mvn
src
main
java
com
projeto
integrado
config
controller
entity
repository
security

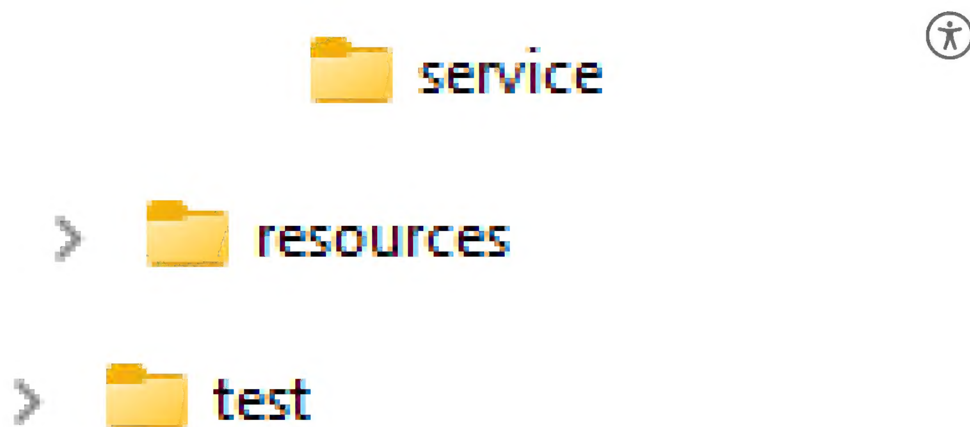


Figura 2: Estrutura de pastas do projeto de backend

Pronto, agora temos, de fato, tudo o que precisamos para começarmos a codificar. Como já mencionado, começaremos pelas entidades. Inclusive, já temos a primeira pronta, que pode ser aproveitada do módulo 5. É a Entidade User. Copie a classe abaixo e a inclua na pasta “entity”.

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/blob/main/modulo5b/codigo_fonte/src/main/java/br/com/descomplica/projeto/seguranca/entity/User.java
(Acesso em 20/10/2022)

Continuando, podemos aproveitar também a entidade Projeto, que criamos no módulo 13, e complementarmos com os atributos e anotações faltantes. Crie uma nova classe, chamada Projeto, dentro de “entity” e cole o código abaixo:

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/blob/main/modulo13/aplicacao/backend/src/m

ain/java/com/projeto/integrado/entity/Projeto.java **(Acesso em 20/10/2022)**



Analisando a tabela projeto e as demais tabelas da aplicação, temos as seguintes relações a envolvendo:

- projeto x tarefa - do tipo 1:N, ou seja, um projeto pode ter várias tabelas; e

- projeto x gerente_projeto - do tipo N:N entre projeto e recurso, sendo refletida no banco pela tabela de ligação gerente_projeto).

Seguindo a lógica acima, ao criarmos a entidade Tarefa, também precisaremos criar as demais entidades com as quais ela se relaciona, ou seja, Recurso (já mencionada em relação a Projeto) e StatusTarefa. Frente a isso, podemos perceber que uma ação leva a outra, num efeito cascata. Logo, é importante aqui tomar cuidado para não deixar nenhuma entidade ou relação de fora. Ao terminar, volte e confira, comparando as entidades e as tabelas, se realizou todos os mapeamentos, incluindo os relacionamentos.

Para ajudá-lo nesse processo de mapeamento, aqui vão algumas dicas, começando pela tabela de ligação. Em nosso modelo físico, temos uma tabela de ligação, a "gerente_projeto". Essa tabela indica existir uma relação N:N entre projeto e recurso. No Spring, os relacionamentos entre tabelas são realizados através de anotações específicas nas instâncias entre as entidades. Sobre esse relacionamento em particular, do tipo "muitos para muitos", há diferentes formas de realizá-lo. Em nosso caso, especificamente, onde a tabela de ligação não possui atributo adicional, mas apenas as chaves estrangeiras das tabelas que ela relaciona, não precisaremos criar uma entidade para a tabela em questão. Com

isso, o relacionamento será feito entre as entidades Projeto e Recurso. Veja o fragmento abaixo, onde esse relacionamento é implementado “dos dois lados”:

```
//Entidade Projeto

@ManyToMany
@JoinTable(
  name = "gerente_projeto",
  joinColumns = @JoinColumn(name = "projeto_id"),
  inverseJoinColumns = @JoinColumn(name = "recurso_id"))
private Set<Recurso> gerentes;



//Entidade Recurso

@ManyToMany(mappedBy = "gerentes")
private Set<Projeto> projetos;
```

Na primeira parte do fragmento acima, temos o mapeamento realizado na entidade Projeto. A primeira anotação utilizada é a @ManyToMany que, como o nome indica, informa que trata-se de uma relação muitos para muitos. A seguir, temos a anotação @JoinTable, onde, em seus parâmetros, é informada a tabela, no banco de dados, responsável pela ligação, assim como as colunas – a joinColumns informa o nome da coluna, no banco de dados, dentro da tabela de ligação, referente à entidade atual/tabela referente à entidade atual, ou seja, projeto. Já na inverseJoinColumns é informada a outra coluna, referente ao outro lado da relação, ou seja, recurso_id da tabela/entidade recurso. Por fim, definimos o tipo de dados como uma coleção do tipo “Set” para a instância da entidade relacionada.

Na segunda parte do fragmento temos o mapeamento da relação sendo feito do outro lado, ou seja, na entidade Recurso. Aqui a anotação é simplificada, envolvendo apenas a @ManyToMany e o parâmetro mappedBy, em que informamos o

nome da instância com a qual a relação foi anotada na entidade Projeto, que, como pode ser visto no primeiro fragmento, chama-se "gerentes".

Seguindo essa mesma linha, a segunda dica também diz respeito a relacionamentos. Depois de falar sobre @ManyToMany, vamos falar dos demais necessários em nosso projeto, onde temos também o @OneToOne, entre Recurso e Tarefa; o @OneToMany entre Projeto e Tarefa / e seu inverso, o @ManyToOne entre Tarefa e Projeto. Nesse ponto cabe um adendo: o Spring trabalha com o conceito de relação bidirecional. Ou seja, quando anotamos a relação dos dois lados, ou seja, nas duas entidades envolvidas, conseguiremos, em termos de operações com essas entidades (como seleção/recuperação de dados) obter os resultados a partir das duas pontas. Para facilitar, vamos exemplificar: ao fazer a anotação de relação @OneToMany entre Projeto/Tarefa e @ManyToOne entre Tarefa/Projeto, conseguiremos obter:

- A partir de uma entidade Projeto, seus dados (representado por seus atributos) e uma lista de todas as tarefas relacionadas a esse projeto;

- A partir de uma entidade Tarefa, seus dados e também os dados do Projeto ao qual a tarefa está relacionada.

Tenha em mente que não é obrigatório anotarmos a relação "dos dois lados". Logo, cada caso deve ser analisado e decidido levando em conta diversos fatores, como necessidade de negócio, real utilização de todos os dados encadeados (já que ligar as entidades dos dois lados aumenta a quantidade de dados trafegados), etc.

A última dica sobre relacionamentos, que é a parte mais sensível no mapeamento objeto relacional, é que o Spring usa um padrão quando realizamos as anotações de forma bi-direcional (nas duas entidades envolvidas). Veja mais esse fragmento, onde consta o relacionamento entre Tarefa e Recurso:

```
//Entidade Tarefa

@OneToOne
@JoinColumn(name = "recurso_id", referencedColumnName = "recurso_id")
Recurso recurso;


//Entidade Recurso

@OneToOne(mappedBy = "recurso")
Tarefa tarefa;
```

Repare que, também nessa relação, um dos lados a anotação de relacionamento simplificada, acompanhada do parâmetro mappedBy, enquanto, no outro lado, temos a anotação da relação acompanhada pela @JoinColumn, contendo como parâmetro as colunas, do banco de dados, responsáveis pelo relacionamento (a chave estrangeira e o seu nome nas duas tabelas envolvidas). Explicando em detalhes:

Na tabela tarefa consta a chave-estrangeira recurso_id. Essa coluna faz referência à tabela recurso, coluna recurso_id. Nesse relacionamento a chave-estrangeira e a chave-primária possuem o mesmo nome. Entretanto, o nome da chave-estrangeira, em algumas situações, por decisão do dev ou DBA, pode receber outro nome. Antes de seguirmos, veja abaixo a instrução SQL que cria essa relação entre as tabelas em questão:

ALTER TABLE tarefa ADD CONSTRAINT fk_tarefa_recurso FOREIGN KEY (recurso_id) REFERENCES recurso(recurso_id);



Agora, repare no fragmento de código acima, que na entidade Tarefa, além do @OneToOne, incluímos também a anotação @JoinColumn, passando os parâmetros name e referencedColumnName. Em name colocamos o nome da chave-estrangeira (a coluna da tabela Tarefa) e em referencedColumnName o nome da chave-primária na tabela relacionada, ou seja, recurso_id .

Para fechar a parte do mapeamento que, inclusive, é a mais complicada de fazer, vou adicionar abaixo como os relacionamentos ficaram em cada Entidade, para que você possa comparar com o seu código.

```
//Relacionamento 1:N com Tarefa
@OneToMany(mappedBy = "projeto")
private Set<Tarefa> tarefas;

//Relacionamento N:N com Recurso
@ManyToMany
@JoinTable(
  name = "gerente_projeto",
  joinColumns = @JoinColumn(name = "projeto_id"),
  inverseJoinColumns = @JoinColumn(name = "recurso_id"))
private Set<Recurso> gerentes;
```

Entidade Projeto



```
    //Relacionamento 1:1 com StatusTarefa
    @OneToOne
    @JoinColumn(name = "status_tarefa_id",
                                referencedColumnName =
          "status_tarefa_id")
    private StatusTarefa statusTarefa;

    //Relacionamento N:1 com Projeto
    @ManyToOne
    @JoinColumn(name = "projeto_id",
                                referencedColumnName =
          "projeto_id")
    Projeto projeto;

    //Relacionamento 1:1 com Recurso
    @OneToOne
    @JoinColumn(name = "recurso_id",
                                referencedColumnName =
          "recurso_id")
    Recurso recurso;
```

Entidade Tarefa

```
    //Relacionamento N:N com Projeto
    @ManyToMany(mappedBy = "gerentes")
    private Set<Projeto> projetos;

    //Relacionamento 1:1 com Tarefa
    @OneToOne(mappedBy = "recurso")
    Tarefa tarefa;
```

Entidade Recurso

```
    //Relacionamento 1:1 com Tarefa
    @OneToOne(mappedBy = "statusTarefa")
    Tarefa tarefa;
```

Entidade StatusTarefa



Continuando no mapeamento e codificação das Entidades, e após termos visto os relacionamentos, temos ainda os demais atributos, cujas únicas particularidades são a utilização da anotação @Column, com o parâmetro name informando o nome da coluna na tabela referente ao atributo em questão. Além disso, merece uma atenção especial o atributo que representa a chave-primária da tabela. Esse atributo recebe algumas anotações extras. Veja abaixo o fragmento da entidade Projeto:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "projeto_id")
private Integer projetoId;
```

Perceba que a primeira anotação é a @Id. Ela é responsável por informar que a coluna em questão faz o papel de chave-primária. Há ainda a anotação @GeneratedValue. Sua função é informar ao Spring qual a estratégia a ser utilizada na geração dos valores desse atributo e de quem é sua responsabilidade. Aqui, vamos recordar que, no banco de dados, uma das restrições para a coluna definida como chave-primária é possuir valores únicos. Frente a isso, os próprios SGBDs têm seus mecanismos para garantir tal restrição, sendo o mais utilizado o auto-incremento. Ou seja, a cada novo registro inserido em uma tabela é incrementado o valor da chave-primária. Aqui no Spring, podemos definir, através da anotação, quem vai ser responsável pela geração desses valores,

se o próprio SGBD, como no exemplo, onde é usado GenerationType.IDENTITY ou não.

Nesse ponto, chegamos ao final da codificação das entidades. Lembre-se de também incluir os métodos Get e Set (lembrando que os atributos devem sempre ser privados, possuindo métodos de acesso get/set públicos). Ao todo, você deve ter 5 entidades criadas ao final desse processo: Projeto, Recurso, StatusTarefa, Tarefa e User.

A etapa seguinte, depois de criadas as entidades, é criarmos os repositórios. Esse processo é simples, visto que o Spring nos permite estender algumas interfaces pré-existentes, que nos fornecem os métodos CRUD de acesso/manipulação de dados que precisamos. Além desses, também podemos criar queries personalizadas. Entretanto, como nosso projeto integrado é um CRUD, não precisaremos, a princípio, criar queries adicionais – exceto para a entidade User (cujos códigos da Entidade, Repositório, Serviço e Controle você pode copiar dos códigos do módulo 5). Veja abaixo o exemplo do repositório ProjetoRepository:

```java
package com.projeto.integrado.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.projeto.integrado.entity.Projeto;

public interface ProjetoRepository
      extends JpaRepository<Projeto, Integer> {}
```

Alguns rápidos comentários sobre o código acima: por convenção, o nome do repositório (do serviço e do controle) são compostos pelo nome da Entidade seguido da palavra Repository (Service, Controller). Repare ainda se tratar de uma interface e não uma

classe Java. Tal interface estende a interface JpaRepository (que fornece os métodos CRUD anteriormente mencionados), sendo possível, a seu critério, estender outras interfaces. Por fim, deve ser informado, como parâmetro ao estender a JpaRepository, o nome da Entidade relacionada ao repositório e o tipo de dados do abritudo @Id da mesma. Em nosso caso, o repositório ProjetoRepository se refere à entidade Projeto, cujo atributo anotado com @Id (chave primária) é do tipo Integer.

Crie todos os repositórios, que totalizam também 5: ProjetoRepository, RecursoRepository, StatusTarefaRepository, TarefaRepository e UserRepository.

Continuando nossa codificação, vamos criar, agora, os serviços. Os serviços são responsáveis pela intermediação entre o controle e o repositório. Nele armazenamos as regras de negócio de nossa aplicação. No nosso caso, implementaremos os métodos CRUD: getAll, getById, save, update e delete. Consequentemente, nossos 5 serviços serão iguais, apenas manipulando, cada um, a respectiva entidade à qual se refere. Lembre-se também de instanciar o repositório, anotando-o com o @Autowired (para que o Spring controle, pra você, a injeção de dependência). Em linhas gerais, cada método que compõe o serviço retornará o respectivo método do repositório. Veja abaixo como ficou o ProjetoService:

```
package com.projeto.integrado.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.projeto.integrado.entity.Projeto;
import com.projeto.integrado.repository.ProjetoRepository;

@Service
public class ProjetoService {
	@Autowired
	ProjetoRepository projetoRepository;

	public List<Projeto> getAll(){
		return projetoRepository.findAll();
	}

	public Projeto getById(Integer id) {
		return projetoRepository.findById(id).orElse(null) ;
	}

	public Projeto saveProjeto(Projeto projeto) {
		return projetoRepository.save(projeto);
	}

	public Projeto updateProjeto(Integer id, Projeto projeto) {
				Projeto projetoAtualizado =
						projetoRepository.findById(id).o
		rElse(null);

	  if(projetoAtualizado != null) {
		projetoAtualizado.setGerentes(projeto.getGerentes());
		projetoAtualizado.setProjetoDescricao(
						projeto.getProjetoDescricao()
				);
		projetoAtualizado.setProjetoFim(projeto.getProjetoFim());
		projetoAtualizado.setProjetoInicio(projeto.getProjetoInicio());
		projetoAtualizado.setProjetoNome(projeto.getProjetoNome());
		projetoAtualizado.setProjetoStatus(projeto.getProjetoStatus());
		return projetoRepository.save(projetoAtualizado);

	  }else {
		return null;
	  }
	}

	public Boolean deleteProjeto(Integer id) {
		Projeto projeto = projetoRepository.findById(id).orElse(null);
		if(projeto != null) {
			projetoRepository.delete(projeto);
			return true;
		}else {
			return false;
		}
	}
```

ProjetoService.java

Crie os demais services em seu código. Lembre-se que tratamos desse assunto no Módulo 4. Logo, em caso de dúvidas, recorra a esse material.

Chegamos à última camada de nossa API, a camada Controller. Nessa camada teremos, a exemplo do que vimos nas demais, um controller para cada entidade. Além disso, teremos um controller para cada serviço implementado (relativos aos métodos CRUD). O cuidado aqui é anotar cada método com o verbo HTTP correto. Veja abaixo como ficou a ProjetoController. A seguir, crie os demais controles.

```
@RestController
@RequestMapping("/projeto")
public class ProjetoController {
	@Autowired
	ProjetoService projetoService;

	@GetMapping
	public ResponseEntity<List<Projeto>> getAll(){
		List<Projeto> projetos = projetoService.getAll();
		if(!projetos.isEmpty())
			return new ResponseEntity<>(projetos, HttpStatus.OK);
		else
			return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
	}

	@GetMapping("/{id}")
	public ResponseEntity<Projeto> getById(@PathVariable Integer id) {
		Projeto projeto = projetoService.getById(id);
		if(projeto != null)
			return new ResponseEntity<>(projeto, HttpStatus.OK);
		else
			return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
	}

	@PostMapping
	public ResponseEntity<Projeto>
				saveProjeto(@RequestBody Projeto projeto) {
		return new ResponseEntity<>
						(projetoService.saveProjeto(projeto),
		HttpStatus.CREATED);
	}

	@PutMapping("/{id}")
	public ResponseEntity<Projeto>
			updateProjeto(@PathVariable Integer id,
						@RequestBody Projeto projeto) {
		Projeto projetoAtualizada =
						projetoService.updateProjeto(id,
		projeto);
		if(projetoAtualizada != null)
			return new
								ResponseEntity<>(pro
			jetoAtualizada, HttpStatus.OK);
		else
			return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
	}

	@DeleteMapping("/{id}")
	public ResponseEntity<Boolean> deleteProjeto(@PathVariable Integer id) {
		if(projetoService.deleteProjeto(id))
			return new ResponseEntity<>(true, HttpStatus.OK);
		else
			return new ResponseEntity<>(false, HttpStatus.OK);
	}
}
```

ProjetoController.java

Pronto, chegamos ao final da codificação, tendo criado as entidades, repositórios, serviços e controles. Entretanto, para finalizar a API, ainda precisaremos finalizar alguns pontos. Como você deve ter percebido, alguns códigos estão apresentando erros – relacionados à entidade User e todo o processo de segurança. Além disso, precisamos configurar a documentação. Para resolver esses pontos, vamos aproveitar alguns códigos vistos em outros módulos. Nesse sentido, comece trazendo, do repositório abaixo, as classes do pacote security.

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo5b/codigo_fonte/src/main/java/br/com/descomplica/projeto/seguranca **(Acesso em 20/10/2022)**

Após trazer os códigos, você precisará realizar alguns ajustes nos scripts importados:

- Alterar o nome do pacote a que pertencem;
- Ajustar os imports, por exemplo, os que importam a entidade User, que nesse projeto está num pacote diferente do que no projeto de origem desses scripts;
- O nome do método, no UserService, responsável por salvar o usuário tem outro nome em nosso projeto. Acerte-o no AuthController.

Além dos ajustes, você precisará também do LoginCredentials. Crie esse package no projeto e traga a classe constante no repositório acima. Ajuste novamente a importação no AuthController.

Para que a API funcione e você consiga testá-la através do Swagger, por exemplo, ou caso queira utilizar a ferramenta Actuator, será necessário liberar as URLs desses serviços na configuração de segurança. Edite a classe SecurityConfig e modifique o método filterChain, conforme abaixo. Aproveitando que precisaremos editar a classe de configuração, acrescente a configuração de header relativa ao cors – ".headers().frameOptions().sameOrigin()". Ao final sua classe deverá ficar assim:

```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception
{
        http.csrf().disable()
                .httpBasic().disable()
                .authorizeHttpRequests()
                .antMatchers("/auth/**",
                     "/swagger-ui/**",
                     "/v3/api-docs/**",
                     "/actuator/**").permitAll()
                .antMatchers("/user/**").hasRole("USER")
                .anyRequest()
                .authenticated()
                .and()
                .userDetailsService(uds)
                .exceptionHandling()
                .authenticationEntryPoint(
                        (request, response, authException) ->

response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized")
                )
                .and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(filter,
UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
```

Classe SecurityConfig – Método filterChain

Atualmente, também temos uma classe, no package config, específica para configuração do CORS. Altere essa classe para que fique assim:

```
@Configuration
public class ConfigurationCors {

	@Bean
	public CorsFilter corsFilter() {
		UrlBasedCorsConfigurationSource source =
							new
		UrlBasedCorsConfigurationSource();
		CorsConfiguration config = new CorsConfiguration();
		config.setAllowCredentials(false);
		config.addAllowedOrigin("*");
		config.addAllowedHeader("*");
		config.addAllowedMethod("GET");
		config.addAllowedMethod("PUT");
		config.addAllowedMethod("POST");
		source.registerCorsConfiguration("/**", config);
		return new CorsFilter(source);
	}
}
```

Classe ConfigurationCors

Nesse caso, como estamos em ambiente de desenvolvimento, permitimos acesso proveniente de qualquer origem, em termos de CORS. Entretanto, ao subir um ambiente de produção, você deverá configurar as opções acima com maior atenção, podendo tal configuração variar caso a caso.

Com isso terminamos os ajustes relativos à segurança. Precisamos agora incluir no package config a configuração do Springdoc, contidos na classe SpringdocConfig, do repositório a seguir:

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo6/src/main/java/br/com/descomplica **(Acesso em 20/10/2022)**

Edite a classe e modifique as informações da API, para que representem os dados de nosso projeto atual ao ex a documentação via Swagger.

Tendo finalizado os últimos ajustes, relativos à segurança e documentação da aplicação, concluímos a sua codificação. Agora, precisaremos testar a API, via Swagger ou algum outro cliente, como o Insomnia. Para isso, você pode usar a estrutura que criamos no módulo 13, quando configurarmos os ambientes dockerizados. Lembre-se de atualizar tanto o código-fonte do backend, quanto o script de criação das tabelas, já que o alteramos para inclusão da tabela user. Faça esses ajustes e suba os contêineres. A seguir, teste cada endpoint da API.

Os códigos da minha implementação e também as coleções de requisições do Insomnia, para testes da API, estão disponíveis no endereço abaixo:

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo14 **(Acesso em 20/10/2022)**

Atividade Extra

Como revisão, recomendo que leia sobre os mapeamentos de relacionamentos entre entidades no Spring nos seguintes links:

https://www.baeldung.com/jpa-many-to-many **(Acesso em 20/10/2022)**

https://www.baeldung.com/hibernate-one-to-many **(Acesso em 20/10/2022)**



Referência Bibliográfica

WALLS, Craig. **Spring Boot in Action.** Shelter Island, NY: Manning Manning, 2016.

Ir para exercício